Review For CA2, Group 4

a) Solve the optimization problem using GD, stochastic GD, SVRG, and SAG;

- In the first part, the group has taken the Transpose of the data and has not explained why this was taken and what are the benefits.
- Division of the data into train and test is hard coded, correct but not efficient
- This is the reason the test data and train data has the following structure :

```
Size of Input data = (5232, 2921)
Size of Input Training data = (5231, 1500)
Size of Output Training data = (1, 1500)
Size of Input Test data = (5231, 1421)
Size of Output Test data = (1, 1421)
```

- The cost function and gradient is implemented correctly.
- The solvers GD, SGD, SVRG, and SAG are implemented
- The results are only provided for three solvers and it is not clear if the last results belong to the SVRG or SAG
- It is shown that SGD has same performance as GD but consumes much less time.
- The comparison is not done and no insight is provided.

b) Part 2 and 3 are not implemented

In [2]:

```
import pandas as pd
import numpy as np
import time
import math
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import resource
from pathlib import Path

giturl1 = 'https://raw.githubusercontent.com/vnmo/MLon/main/CA1_1c_greenhouse_cleanedDat
giturl2 = 'https://raw.githubusercontent.com/vnmo/MLon/main/CA1_1c_greenhouse_cleanedDat
data1  = pd.read_csv(giturl1)
data2  = pd.read_csv(giturl2)
data_complete = pd.concat([data1,data2],ignore_index=True)
```

Nicely extracted the data from the given file. It was a confusing task but managed well by analyzing the data in matlab and saving them separately.

In [3]:

```
data=data_complete.T
print(f"Size of Input data = {data.shape}\n")
data=data.to_numpy()

X_train = data[0:5231,0:1500]
X_test = data[0:5231,1500:2921]

Y_train = data[5231:5232,0:1500]
Y_test = data[5231:5232,1500:2921]

# X_train=X_train.to_numpy()
# X_test=X_test.to_numpy()
# Y_train=Y_train.to_numpy()
# Y_test=Y_test.to_numpy()

print(f"Size of Input Training data = {X_train.shape}")
print(f"Size of Output Training data = {Y_train.shape}")
print(f"Size of Input Test data = {X_test.shape}")
print(f"Size of Output Test data = {Y_test.shape}")
```

```
Size of Input data = (5232, 2921)

Size of Input Training data = (5231, 1500)
Size of Output Training data = (1, 1500)
Size of Input Test data = (5231, 1421)
Size of Output Test data = (1, 1421)
```

Calculating the cost function using: $f_i(w) = log(1 + \exp(-y_i w^{\top} x_i)) + 2 * ||w||^2$

Calculating the gradient of $f_i(w)$ using:

$$\nabla f_i = (\frac{1}{N} \sum_{i \in [N]} \frac{-y_i * e^{-y_i w^T x_i}}{e^{-y_i w^T x_i} + 1} x_i + 2\lambda w)$$

Calculation of gradient is correct

In [4]:

```python
## Logistic ridge regression with different optimizers
# cost function and gradient calculation

def cost(x,y,w,lambda_):
    #Send in properly shaped NUMPY arrays

    D, N = x.shape
    dy,ny = y.shape
    dw,nw = w.shape
```

Unused variables here in this function

```python
    value = 0
    for i in range(N):
        exponent = np.exp(-y[0,i]*(w.T@x[:,i]))
        value += np.log(1+exponent)
    c = lambda_ * (w.T@w)
    return value/N + c

def function_gradient(x, y, w, lambda_):
    #Send in properly shaped NUMPY arrays
    D, N = x.shape
    dy,ny = y.shape
    dw,nw = w.shape

    if (dw!=D or nw!=1):
      print("Dimension of w is not correct")
      return
```

Correct implementation

```python
    if (dy!=1 or ny!=N):
      print("Dimension of y is not correct")
      return


    value = np.zeros((D, 1))
    for i in range(N):
      exponent = np.exp(-y[0,i]*(w.T@x[:,i]))
      increment= (-1*y[0,i]) * (exponent/(1+exponent)) * (x[:,i]);
      value += increment.reshape(D, 1)
    delF = value/N + 2* lambda_ * w
    return delF


w = X_train[:,0:1]; ## TESTING with random w
lambda_= 1
c = cost(X_test,Y_test,w,lambda_)
delF = function_gradient(X_test, Y_test, w, lambda_)
print(c)
print(delF)


# print(c)
#D = Y_train.shape
#print(D)
```

```
[[7336230.67442909]]
[[2.4062520e-04]
 [2.4992540e-04]
 [2.4160600e-04]
 ...
 [4.8844600e-02]
 [1.6970608e+02]
 [7.6191000e+00]]
```

Good practice to analyze the outputs

In [5]:

```
## Define solvers: GD, SGD, SVRG and SAG.
# Setting the values here:

alpha = 0.5
num_iters = 25
lambda_ = 0.01
epsilon = 0.0001
# ---------------------- Complete the blank definitions: ------------------------------

def solver(x,y, w, alpha, num_iters , lambda_ , epsilon , optimizer = "GD",mem=False):

    D, N = x.shape
    dy,ny=y.shape
    dw,nw=w.shape

    if (dw!=D or nw!=1):
      print("Dimension of w is not correct")
      return
    if (dy!=1 or ny!=N):
      print("Dimension of y is not correct")
      return



    if (optimizer == "GD") :
        for i in range(num_iters):
            # update the parameter w for GD here:

            g = function_gradient(x, y, w, lambda_)
            w = w - (alpha*g)

            if (i%10==0) and (mem):
                usage=resource.getrusage(resource.RUSAGE_SELF)
                print("mem for GD (mb):", (usage[2]*resource.getpagesize())/1000000.0)
            if np.linalg.norm(g) <= epsilon:
                break
            #print(w)

    elif (optimizer == "SGD"):
        for i in range(num_iters):
            # Complete SGD here:
            Num_Sam = int(np.random.rand() * x.shape[1])
            while Num_Sam == 0:
              Num_Sam = int(np.random.rand() * x.shape[1])

            ind = np.arange(x.shape[1])[:Num_Sam]
            g = function_gradient(x[:, ind], y[:, ind], w, lambda_)
            w = w - alpha*g

            if (i%100==0) and (mem):
                usage=resource.getrusage(resource.RUSAGE_SELF)
                loss = cost(x,y,w,lambda_)
                print(i, loss)
            if (np.linalg.norm(g) <= epsilon):
                break


    elif (optimizer == "SVRG"):
        i = 0
```

Correct!!

Based on pseudo algorithm, logic seems to be correct for all the three algo

```
    J = 200
    K = 50
    for k in range(K):

        g = function_gradient(x, y, w, lambda_)
        w_old = w

        for j in range(J):
          Num_Sam = int(np.random.rand() * x.shape[1])
          while Num_Sam == 0:
            Num_Sam = int(np.random.rand() * x.shape[1])

          ind = np.arange(x.shape[1])
          np.random.shuffle(ind)
          ind = ind[:Num_Sam]

          stepA = function_gradient(x[:, ind], y[:, ind], w_old, lambda_)
          stepB = function_gradient(x[:, ind], y[:, ind], w, lambda_)
          stepC = g

          w_old = w_old - alpha*(stepA - stepB + stepC)

        w = w_old
        i = i+1

        if (i%100==0) and (mem):
                usage=resource.getrusage(resource.RUSAGE_SELF)
                loss = cost(x,y,w,lambda_)
                print(i, loss)
        if (np.linalg.norm((stepA - stepB + stepC)) <= epsilon):
            break


elif (optimizer == "SAG"):
    Num_Sam = 5231
    dw = np.zeros(w.shape)
    gi = np.zeros(N, dy)


    for i in range(num_iters):
        ind = np.arange(x.shape[1])
        np.random.shuffle(ind)
        comb_ind = ind[:Num_Sam]

        g = function_gradient(x[:, comb_ind].reshape(D, comb_ind.size), y[:, comb_in
        print(f"g = {g.shape}")
        print(f"gi = {gi.shape}")
        gi[comb_ind,:] = g
        print(f"gi = {gi.shape}")

        dw = np.sum(gi[:,:],axis=0)
        w = w - alpha * dw/N
        if (np.linalg.norm(dw/N) <= epsilon):
            break

        if (i%100==0) and (mem):
            loss = cost(x,y,w,lambda_)
            usage=resource.getrusage(resource.RUSAGE_SELF)
            print(i, loss)
    i=k
```

```
    return w
```

In [6]:

```
## Solving the optimization problem:

#y = np.array(Y_train.iloc[0:6000])
#x = np.array(X_train.iloc[0:6000,:])

#y = np.array(Y_train.iloc[0:6000])
#x = np.array(X_train.iloc[0:6000,:])
D,N = X_train.shape
w = np.random.rand(D,1)*0.01

D, N = X_train.shape
d = 1
#w = np.random.normal(0,0.1, D*d).reshape(D,d)
#w = np.random.rand(D,1)*0.01
print(w.shape) # Initialization of w

#-------------------- GD Solver -----------------------
start=time.time()
gde = solver(X_train, Y_train, w, alpha, num_iters, lambda_, epsilon, optimizer="GD", me
end = time.time()
print("Weights of GD after convergence: \n",gde)

cost_value = cost(X_test,Y_test,gde,lambda_)  # Calculate the cost value
print("Cost of GD after convergence: ",cost_value)
print("Training time for GD: ", end-start , ' seconds')




#-------------------- SGD Solver -----------------------
start=time.time()
sgde = solver(X_train, Y_train, w, alpha, num_iters, lambda_, epsilon, optimizer="SGD",
end = time.time()
print("Weights of SGD after convergence: \n",sgde)

cost_value = cost(X_test,Y_test,sgde,lambda_)  # Calculate the cost value
print("Cost of SGD after convergence: ",cost_value)
print("Training time for SGD: ", end-start , ' seconds')



#-------------------- SVRG Solver -----------------------
start=time.time()
svrg = solver(X_train, Y_train, w, alpha, num_iters, lambda_, epsilon, optimizer="SGD",
end = time.time()
print("Weights of SVRG after convergence: \n",svrg)

cost_value = cost(X_test,Y_test,svrg,lambda_)  # Calculate the cost value
print("Cost of SVRG after convergence: ",cost_value)
print("Training time for SVRG: ", end-start , ' seconds')





#-------------------- SAG Solver -----------------------
start=time.time()
sag = solver(X_train, Y_train, w, alpha, num_iters, lambda_, epsilon, optimizer="SGD", m
end = time.time()
```

```
print("Weights of SAG after convergence: \n",sag)

cost_value = cost(X_test,Y_test,sag,lambda_)  # Calculate the cost value
print("Cost of SAG after convergence: ",cost_value)
print("Training time for SAG: ", end-start , ' seconds')
```

```
(5231, 1)
Weights of GD after convergence:
 [[0.00765498]
 [0.00579736]
 [0.00047369]
 ...
 [0.00390197]
 [0.00334771]
 [0.00013224]]
Cost of GD after convergence:  [[0.00108435]]
Training time for GD:  4.373081207275391  seconds
Weights of SGD after convergence:
 [[0.00765498]
 [0.00579736]
 [0.00047369]
 ...
 [0.00391442]
 [0.00335827]
 [0.00016261]]
Cost of SGD after convergence:  [[0.00111445]]
Training time for SGD:  1.089224100112915  seconds
Weights of SVRG after convergence:
 [[0.00765499]
 [0.00579736]
 [0.00047369]
 ...
 [0.00392598]
 [0.00336806]
 [0.00019077]]
Cost of SVRG after convergence:  [[0.00114543]]
Training time for SVRG:  1.2613723278045654  seconds
Weights of SAG after convergence:
 [[0.00765498]
 [0.00579736]
 [0.00047369]
 ...
 [0.00391798]
 [0.00336128]
 [0.00017127]]
Cost of SAG after convergence:  [[0.00112367]]
Training time for SAG:  1.1732664108276367  seconds
```

Would have been better if we had some visualization of these performance indicators wrt different parameters. It will be nice to see those graphs and analyze their characteristics.